**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

по дисциплине: «Распределенные информационные системы»

на тему: **«**Тупики. Предотвращение тупиковых ситуаций. Синхронизация процессов**»**

Выполнил: студент гр. ИТИ-41

Ковшаров Г. Ю.

Принял: преподаватель

Башаримов Ю. С.

Гомель 2025

**Цель работы:** изучить практическое применение многопоточного программирования для решения вычислительно сложной геометрической задачи, освоить методику синхронизации потоков с помощью флагов готовности и проанализировать эффективность распараллеливания алгоритма путем сравнения времени выполнения однопоточной и многопоточной версий программы.

**Задание:**

Необходимо разработать однопоточную и многопоточную программы, осуществляющие решение поставленной задачи (таблица 1). В многопоточном приложении синхронизацию потоков выполнить указанным в таблице 1 способом. Приложение должно позволять вводить количество используемых потоков для вычислений (1-4). Необходимо замерять время решения задачи при использовании различного числа потоков, сравнить время вычислений и результаты решения, сделать выводы Исходные данные для решения задачи разместить в текстовом файле.

Таблица 1 – Вариант задания

|  |  |  |  |
| --- | --- | --- | --- |
| № | Условие задачи | Действия, выполняемые в потоках | Метод синхронизации |
| 13 | Текстовый файл содержит координаты точек в трёхмерном пространстве (*N*=1000). Необходимо определить площади всех возможных трапеций, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь. | 1.Формирование массива точек, которые могут быть вершинами одной трапеции  2. Вычисление площади и углов трапеции | Флаги готовности |

**Ход работы**

Напишем код на языке *C++*. Основной алгоритм поиска трапеций основан на переборе всех уникальных комбинаций из четырех точек. Для каждой такой комбинации выполняется проверка на компланарность (принадлежность одной плоскости), а затем на параллельность одной из трех возможных пар противоположных сторон.

Файл с точками *points* имеет вид, представленный на рисунке 1.

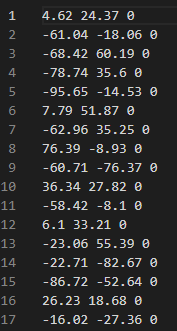


Рисунок 1 – Скриншот файла с точками

В однопоточной версии этот перебор выполняется последовательно в главном потоке программы.

В многопоточной версии основной цикл перебора точек распределяется между потоками по принципу «каждый *N*-й индекс**»**, где *N* – количество потоков. Каждый поток выполняет поиск трапеций для своей части индексов. Для организации взаимного исключения при записи в общий вектор результатов используется массив атомарных флагов, которые реализует механизм «флагов готовности**»**.

Перед добавлением локальных результатов в общий вектор поток проверяет флаги других потоков и ожидает, пока они не выйдут из критической секции. После завершения записи поток сбрасывает свой флаг. Главный поток запускает все рабочие потоки и ожидает их завершения. Такой подход позволяет избежать гонок за общий ресурс и эффективно синхронизирует потоки без использования стандартного *mutex*/*lock\_guard*.

Для оценки производительности были проведены замеры времени выполнения программы на наборе данных из 500 точек при разном количестве задействованных потоков.

Таблица 2 – Таблица сравнения решений для 500 точек

|  |  |
| --- | --- |
| Кол-во потоков | Время выполнения (сек) |
| 1 | 220.651 |
| 2 | 93.6991 |
| 3 | 75.58 |
| 4 | 56.1827 |

Анализ результатов показывает, что распараллеливание задачи дает существенное ускорение. Переход от одного потока к двум сокращает время выполнения в два с половиной раза. Дальнейшее увеличение числа потоков также повышает производительность.

На рисунке 2 продемонстрирована работа разработанного консольного приложения.

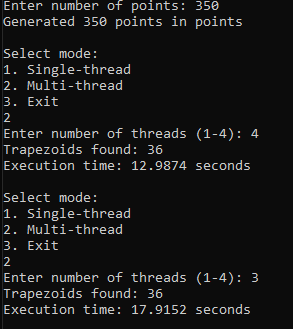


Рисунок 2 – Скриншот вывода результатов в консоль после   
работы приложения

**Вывод:** в результате выполнения лабораторной работы была успешно разработана и протестирована программа для поиска трапеций в трехмерном пространстве в однопоточном и многопоточном режимах, для многопоточного режима был реализован метод синхронизации – Флаги готовности, чья эффективность и корректность была проверена. Было продемонстрировано, что для вычислительно интенсивных задач с возможностью параллелизации данных многопоточный подход позволяет значительно сократить время выполнения.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг программы**

main.cpp  
#include "point3d.h"

#include "math\_utils.h"

#include "file\_utils.h"

#include "thread\_utils.h"

#include <iostream>

#include <vector>

using namespace std;

int main() {

    const string filename = "points";

    int N\_POINTS = 0;

    while(N\_POINTS <= 0) {

        cout << "Enter number of points: ";

        cin >> N\_POINTS;

    }

    generate\_data\_file(filename, N\_POINTS);

    vector<Point3D> points = read\_points\_from\_file(filename);

    if(points.empty()) return 1;

    int choice = 0;

    while(choice != 3) {

        cout << "\nSelect mode:\n1. Single-thread\n2. Multi-thread\n3. Exit\n";

        cin >> choice;

        switch(choice) {

            case 1: run\_single\_threaded(points); break;

            case 2: {

                int num\_threads = 0;

                while(num\_threads<1||num\_threads>4) {

                    cout << "Enter number of threads (1-4): ";

                    cin >> num\_threads;

                }

                run\_multi\_thread(points,num\_threads);

                break;

            }

            default: cout << "Invalid choice\n";

        }

    }

    return 0;

}

file\_utils.h  
#pragma once

#include "point3d.h"

#include <string>

#include <vector>

using namespace std;

void generate\_data\_file(const string& filename, int n);

vector<Point3D> read\_points\_from\_file(const string& filename);

void write\_results\_to\_file(const string& filename, const vector<TrapezoidResult>& results);

file\_utils.cpp  
#include "file\_utils.h"

#include <fstream>

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <iomanip>

void generate\_data\_file(const string& filename, int n) {

ofstream file(filename);

if (!file) { cerr << "Cannot create file\n"; return; }

srand((unsigned)time(nullptr));

for (int i = 0; i < n; ++i) {

double x = (rand()%20001 - 10000)/100.0;

double y = (rand()%20001 - 10000)/100.0;

double z = 0.0;

file << x << " " << y << " " << z << "\n";

}

cout << "Generated " << n << " points in " << filename << "\n";

}

vector<Point3D> read\_points\_from\_file(const string& filename) {

ifstream file(filename);

vector<Point3D> points;

Point3D p;

while(file >> p.x >> p.y >> p.z) points.push\_back(p);

return points;

}

void write\_results\_to\_file(const string& filename, const vector<TrapezoidResult>& results) {

ofstream file(filename);

if (!file) { cerr << "Cannot write file\n"; return; }

file << fixed << setprecision(4);

for (auto& res : results) {

file << "Vertices: ";

for (auto& v : res.vertices)

file << "{" << v.x << ", " << v.y << ", " << v.z << "} ";

file << "\nAngles: ";

for (auto& a : res.angles) file << a << "° ";

file << "\nArea: " << res.area << "\n\n";

}

}

math\_utils.h  
#pragma once

#include "point3d.h"

#include <vector>

#include <array>

constexpr double EPSILON = 1e-9;

constexpr double PI = 3.14159265358979323846;

using Vector3D = Point3D;

Vector3D operator-(const Point3D& a, const Point3D& b);

double dot\_product(const Vector3D& v1, const Vector3D& v2);

Vector3D cross\_product(const Vector3D& v1, const Vector3D& v2);

double magnitude(const Vector3D& v);

bool are\_parallel(const Vector3D& v1, const Vector3D& v2);

double clamp(double v, double lo = -1.0, double hi = 1.0);

double calculate\_angle(const Point3D& p\_prev, const Point3D& p\_curr, const Point3D& p\_next);

bool are\_coplanar(const Point3D& p1, const Point3D& p2, const Point3D& p3, const Point3D& p4);

void process\_combination(const std::array<Point3D, 4>& points, std::vector<TrapezoidResult>& local\_results);

math\_utils.cpp  
#include "math\_utils.h"

#include <cmath>

#include <iostream>

#include <chrono>

#include <iomanip>

Vector3D operator-(const Point3D& a, const Point3D& b) {

return {a.x - b.x, a.y - b.y, a.z - b.z};

}

double dot\_product(const Vector3D& v1, const Vector3D& v2) {

return v1.x\*v2.x + v1.y\*v2.y + v1.z\*v2.z;

}

Vector3D cross\_product(const Vector3D& v1, const Vector3D& v2) {

return {

v1.y\*v2.z - v1.z\*v2.y,

v1.z\*v2.x - v1.x\*v2.z,

v1.x\*v2.y - v1.y\*v2.x

};

}

double magnitude(const Vector3D& v) {

return sqrt(dot\_product(v, v));

}

bool are\_parallel(const Vector3D& v1, const Vector3D& v2) {

return magnitude(cross\_product(v1, v2)) < EPSILON;

}

double clamp(double v, double lo, double hi) {

if (v < lo) return lo;

if (v > hi) return hi;

return v;

}

double calculate\_angle(const Point3D& p\_prev, const Point3D& p\_curr, const Point3D& p\_next) {

Vector3D v1 = p\_prev - p\_curr;

Vector3D v2 = p\_next - p\_curr;

double dot = dot\_product(v1, v2);

double mag1 = magnitude(v1);

double mag2 = magnitude(v2);

if (mag1 < EPSILON || mag2 < EPSILON) return 0.0;

double cosv = dot / (mag1 \* mag2);

cosv = clamp(cosv, -1.0, 1.0);

return acos(cosv) \* 180.0 / PI;

}

bool are\_coplanar(const Point3D& p1, const Point3D& p2, const Point3D& p3, const Point3D& p4) {

Vector3D v1 = p2 - p1;

Vector3D v2 = p3 - p1;

Vector3D v3 = p4 - p1;

double triple = dot\_product(v1, cross\_product(v2, v3));

double scale = magnitude(v1) \* magnitude(cross\_product(v2, v3));

if (scale < EPSILON) return true;

return fabs(triple) <= 1e-9 \* scale;

}

void process\_combination(const std::array<Point3D, 4>& points, std::vector<TrapezoidResult>& local\_results) {

const Point3D& p1 = points[0];

const Point3D& p2 = points[1];

const Point3D& p3 = points[2];

const Point3D& p4 = points[3];

if (!are\_coplanar(p1, p2, p3, p4)) return;

Vector3D v12 = p2 - p1;

Vector3D v34 = p4 - p3;

Vector3D v13 = p3 - p1;

Vector3D v24 = p4 - p2;

Vector3D v14 = p4 - p1;

Vector3D v23 = p3 - p2;

std::array<Point3D, 4> ordered\_vertices;

bool is\_trapezoid = false;

if (are\_parallel(v12, v34) && magnitude(v13) > EPSILON && magnitude(v24) > EPSILON) {

ordered\_vertices = {p1, p2, p4, p3};

is\_trapezoid = true;

} else if (are\_parallel(v13, v24) && magnitude(v12) > EPSILON && magnitude(v34) > EPSILON) {

ordered\_vertices = {p1, p3, p4, p2};

is\_trapezoid = true;

} else if (are\_parallel(v14, v23) && magnitude(v12) > EPSILON && magnitude(v34) > EPSILON) {

ordered\_vertices = {p1, p4, p3, p2};

is\_trapezoid = true;

}

if (!is\_trapezoid) return;

TrapezoidResult res;

res.vertices = ordered\_vertices;

Vector3D d1 = ordered\_vertices[2] - ordered\_vertices[0];

Vector3D d2 = ordered\_vertices[1] - ordered\_vertices[0];

Vector3D d3 = ordered\_vertices[3] - ordered\_vertices[0];

res.area = 0.5\*magnitude(cross\_product(d2, d1)) + 0.5\*magnitude(cross\_product(d1, d3));

res.angles[0] = calculate\_angle(ordered\_vertices[3], ordered\_vertices[0], ordered\_vertices[1]);

res.angles[1] = calculate\_angle(ordered\_vertices[0], ordered\_vertices[1], ordered\_vertices[2]);

res.angles[2] = calculate\_angle(ordered\_vertices[1], ordered\_vertices[2], ordered\_vertices[3]);

res.angles[3] = calculate\_angle(ordered\_vertices[2], ordered\_vertices[3], ordered\_vertices[0]);

double angle\_sum = res.angles[0] + res.angles[1] + res.angles[2] + res.angles[3];

if (res.area > EPSILON && fabs(angle\_sum - 360.0) < 1.0) {

local\_results.push\_back(res);

}

}

point3d.h  
#pragma once

#include <array>

#include <vector>

#include <atomic>

struct Point3D {

double x, y, z;

};

struct TrapezoidResult {

std::array<Point3D, 4> vertices;

double area;

std::array<double, 4> angles;

};

struct ThreadData {

const std::vector<Point3D>\* points;

std::vector<TrapezoidResult>\* shared\_results;

std::atomic<bool>\* ready\_flags;

int thread\_id;

int num\_threads;

};

thread\_utils.h  
#pragma once

#define WIN32\_LEAN\_AND\_MEAN

#include <cstddef>

#ifdef byte

#undef byte

#endif

#include <windows.h>

#include "point3d.h"

#include <vector>

#include <atomic>

using namespace std;

DWORD WINAPI worker\_with\_flags(LPVOID lpParam);

void run\_single\_threaded(const std::vector<Point3D>& points);

void run\_multi\_thread(const std::vector<Point3D>& points, int num\_threads);

thread\_utils.cpp  
#include "thread\_utils.h"

#include "math\_utils.h"

#include "file\_utils.h"

#include <windows.h>

#include <vector>

#include <iostream>

#include <chrono>

DWORD WINAPI worker\_with\_flags(LPVOID lpParam) {

ThreadData\* data = (ThreadData\*)lpParam;

const vector<Point3D>& points = \*data->points;

vector<TrapezoidResult>& shared\_results = \*data->shared\_results;

atomic<bool>\* ready\_flags = data->ready\_flags;

int thread\_id = data->thread\_id;

int num\_threads = data->num\_threads;

int n = (int)points.size();

for (int i = thread\_id; i < n-3; i += num\_threads) {

vector<TrapezoidResult> local\_results;

for (int j = i+1; j<n-2; ++j)

for (int k=j+1; k<n-1; ++k)

for (int l=k+1; l<n; ++l)

process\_combination({points[i], points[j], points[k], points[l]}, local\_results);

ready\_flags[thread\_id].store(true);

for (int other=0; other<num\_threads; ++other)

if (other!=thread\_id)

while(ready\_flags[other].load())

Sleep(1);

shared\_results.insert(shared\_results.end(), local\_results.begin(), local\_results.end());

ready\_flags[thread\_id].store(false);

}

return 0;

}

void run\_single\_threaded(const vector<Point3D>& points) {

auto start = chrono::high\_resolution\_clock::now();

vector<TrapezoidResult> results;

size\_t n = points.size();

for (size\_t i = 0; i < n; ++i)

for (size\_t j = i + 1; j < n; ++j)

for (size\_t k = j + 1; k < n; ++k)

for (size\_t l = k + 1; l < n; ++l)

process\_combination({points[i], points[j], points[k], points[l]}, results);

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

write\_results\_to\_file("single\_thread", results);

cout << "Trapezoids found: " << results.size() << endl;

cout << "Execution time: " << duration.count() << " seconds" << endl;

}

void run\_multi\_thread(const vector<Point3D>& points, int num\_threads) {

auto start = chrono::high\_resolution\_clock::now();

vector<TrapezoidResult> shared\_results;

vector<HANDLE> threads(num\_threads);

vector<ThreadData> thread\_data(num\_threads);

vector<atomic<bool>> ready\_flags(num\_threads);

for (int i = 0; i < num\_threads; ++i) ready\_flags[i].store(false);

for (int i = 0; i < num\_threads; ++i) {

thread\_data[i].points = &points;

thread\_data[i].shared\_results = &shared\_results;

thread\_data[i].ready\_flags = ready\_flags.data();

thread\_data[i].thread\_id = i;

thread\_data[i].num\_threads = num\_threads;

threads[i] = CreateThread(

nullptr,

0,

worker\_with\_flags,

&thread\_data[i],

0,

nullptr

);

}

WaitForMultipleObjects(num\_threads, threads.data(), TRUE, INFINITE);

for (int i = 0; i < num\_threads; ++i) CloseHandle(threads[i]);

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

write\_results\_to\_file("multi\_thread", shared\_results);

cout << "Trapezoids found: " << shared\_results.size() << "\n";

cout << "Execution time: " << duration.count() << " seconds\n";

}